EN.605.202.81: Introduction to Data Structures

Peter Sullivan

Lab 2 Analysis

Due Date: 7/23/24

Lab 2 Analysis

Data Structures

The lab assignment requested that we set up a process for maneuvering from a prefix expression to a postfix expression. There are many ways that this could be implemented. In lab 1, we used a stack implementation to iteratively process each character for converting from prefix to postfix. For this lab, we are processing a postfix expression using a recursive approach.

In this lab, I created two classes, the expression and operands class. The Expression class is the meat of the program, where we can find the recursive functions that enables users to convert from Postfix to Prefix and Infix, and Prefix to Postfix and Infix. The expressions class utilizes one additional class, the operand class which combines strings together based on the desired input and output. For example, going from a prefix to a postfix expression. To loop through the prefix and postfix strings, I use recursive methods in the expression class. To keep track of the current desired expression string, I use an array list to handle each character. Recursively processing the string using an array list allows me to appropriately keep track and combine operands and operators together when converting to postfix or any other desired expression. The array list data structure is great for this process, due to the speed of accessing elements in the data structure. Adding and pulling elements will give us O(1) complexity. The program also grabs characters in the string using indexing, which is also a 0(1) complexity.

Data Structure Justification/Implementation

Before choosing data types and structures, I first needed to understand how to convert from prefix to postfix. When converting from prefix to post fix, the first step is to read in the expression from right to left. This really means we're reading in the string in reverse. While reading in the string we are keeping track of operands and keeping track of operators. When an operator comes up, this is when we take two of the most recent items that were processed, and we join those into a new operand.

So, we need to read in the data in reverse, and then we need to keep track of recent items. An array data structure is a perfect data structure for this application. First reversing a string is quite easy with a recursive method. You grab the last element of the string using index slicing and push that to the front and reprocess the string again with out the last element. You complete this until each element is processed and then the reverse string is returned. For the process of converting from prefix to postfix, we can use an array data structure for storing the output expression between recursive calls, specifically a non-linked list to keep track of the converted expression as we process the expression. We will process through the expression recursively keeping track of the index, pushing the current element for the recursive call to the array. We continue this process until there are no more characters in the prefix expression. As we process and combine each character, we should end up with only one single item in the array, which should be the postfix representation of the initial prefix expression.

Another option would be using a deque, though we would have to make sure that we are grabbing the most recent instead of the first in first out method that a deque offers. We could also use a stack as our previous assignment, and we could process expression iteratively instead of recursively.

Complexity

Recursive Implementation
In order to figure out the complexity of this program, we can look at the pre_to_post() method in the expressions class. This recursive method will take in a string and will traverse through that string while incrementing the index and storing a list of strings in an array.

When walking through the complexity of the analysis, most of the calls are 0(1). For example, any of the if clauses, grabbing a character in a string using the index, and appending/popping from the array. Before we grab the character at a specific index, we do reverse the string using the reverse order method. That method methodically goes through the string by index, which makes the complexity O(N).

For the main pre_to_post() method, we also do process the string for every component, calling the recursive method each time. The time complexity of the pre_to_post function would also be O(n). The space complexity would also be the space as the time complexity. We are creating an array, but that array will only be as large as the string processed. We do have the nested reverse order method in the pre_to_post call, but that method would only be added to the O(N) call as well. So even if we call the reverse method every time the function runs, the constant multiplied against O(N) will be dropped when N becomes large. For example, 5O(1) + 7O(N), will become O(N) time complexity.

For the recursive solution, both time and space complexity will be O(N).

Stack Implementation
My previous solution for the lab was using a stack implementation. The stack implementation had the same complexity as the current recursive solution. The stack solution had a time and space complexity of O(N). The stack implementation was much more complicated than when compared to the recursive solution. In order to use the stack implementation, we needed to create a stack class which took a lot more coding then when compared to the recursive solution.

The recursive solution was also much easier to comprehend then the stack implementation, due to the minimal amount of coding required. Instead of having a for or while loop used in the iterative approach, the recursive solution would methodically process each character during the call. I found this very convenient during initial debugging. Instead of having to trace errors back to the stack class, I could put print statements or exceptions directly in the recursive method.

In the previous lab, I did choose the recursive solution over stack method. Previously, I mentioned the main reason being that the recursive solution would be simpler. That is still the case for me. After actually working on the recursive solution, I'm also swaying more to the recursive solution due to the ease of debugging. If anything, I am more swayed towards the recursive solution then before during lab 1.

Enhancements

The original lab asked to only convert from prefix to postfix. For this lab I added additional functionality to convert from Prefix to Postfix, Prefix to Infix, Postfix to Prefix, Postfix to Infix. Each method found in the expressions class uses a similar recursive method as the main pre_to_post() method.

To minimize user input, I added in helper functions with the expression class to pick the correct method when running. For example, the user would only need to enter Postfix Infix, and the program will automatically pick the correct method to perform that option.

I also added in two additional arguments that the user can enter if desired from the command line. The default if the two additional are not entered is to run from prefix to postfix. But the user has the option to add two additional arguments at the end of the cmd line, input_type and output_type. The following works:

- python -m lab1 resources\Required_Input.txt output/output.txt
- python -m lab1 resources\Required_Input.txt output/output.txt Prefix Postfix
- python -m lab1 resources\Required_Input.txt output/output.txt Postfix Prefix
- python -m lab1 resources\Required_Input.txt output/output.txt Prefix Infix
- python -m lab1 resources\Required_Input.txt output/output.txt Postfix Infix

The two arguments can be adjusted based on the desired output and the input files provided. Depending on the input type and output type selected, this will tell the expression object which method to run in order to correctly parse and convert from the original input type to the desired output type.

Lessons Learned

If I were going to do this project again, I would try processing the expression and adding to a string instead of adding the processed characters or operands to an array. I believe the process itself would work, but I would lose the ability of some of my error checking. The array data structure had some convenience when checking if the number of operators were correct or incorrect. It was as simple as checking if there were more than one element in the array after processing the entire string. This is not a game ender, but I would have to find a new way to validate my processed expressions.

I'm still not as familiar with modules as I would like to be. Luckily, we do have 2 more labs for me to work on this. I am currently using vs code for testing and project creation, which is causing some difficulty for me with testing when setting up my notebooks. For the module, I understand where to set up my locations for importing other classes that I've created. So when testing via the cmd line, the process works fine. This is what is expected for the project.

But when I want to test running via a juptyer notebook in vs code, I have to modify the locations of the .py files, since vs code does not open up the notebook in the location of the module (where I'm running the cmd line). A fix to this could be changing the path of the current project to the lab location. I will give this a try during lab 3.

Overall, I enjoyed this lab! I used to find recursive functions very difficult to understand. But this process has really solidified my understanding.